Tokens autocontidos com JWT

Ambas abordagens que conhecemos até agora para validação de tokens possuem desvantagens que podem onerar o banco de dados ou a rede. Existe algum tipo de alternativa na qual o Resource Server não precise perguntar nada para o Authorization Server a respeito da validade de um token? E se o próprio token pudesse ser gerado a partir dos dados que precisamos, como o e-mail do Resource Owner e o período de validade do token?

É exatamente isso que a especificação JWT (https://jwt.io/introduction) oferece como solução. Através de um JWT, podemos transportar dados em formato JSON de maneira compactada através de uma string codificada em Base 64. Quais dados são suficientes para identificar um token de acesso? O que são esses dados? JWT define cada um dos atributos que identificam um token como *claims*.

> **IDENTIDADE BASEADA EM CLAIMS**
>
> Uma *claim* nada mais é do que uma expressão que define o que um determinado objeto é. Tenha como objeto qualquer entidade que possa ser identificada como uma pessoa ou um sistema. Para facilitar o entendimento sobre o significado de *claim*, traduzindo essa palavra para o português, podemos interpretar como uma reivindicação ou afirmação que um objeto faz sobre si mesmo ou sobre outro objeto.

Para identificar um token, basicamente precisamos de dados como o tempo de expiração do token, a audiência (em qual Resource Server deve ser usado), quem autorizou a geração do token ou até mesmo quem gerou o token (no caso um identificador do Authorization Server que gerou o token). A especificação JWT classifica as *claims* em três:

- registered: são as *claims* definidas na própria especificação conforme veremos logo adiante. Os atributos **registrados** que identificam um token JWT, segundo a especificação, são os seguintes: iss, sub, aud, exp, nbf, iat e jti (não se preocupe com esses nomes agora, pois veremos a definição de cada um logo a seguir). As *claims* registradas não são obrigatórias, porém, pelo falto de serem padronizadas, ajudam a manter a interoperabilidade entre sistemas que usam JWT.
- public: são atributos que podem ser definidos por uma aplicação e que podem ser usados fora do ambiente dessa aplicação. Como se tratam de atributos públicos, é interessante que estes sejam registrados para evitar colisão de nomes. O processo para registrar um novo *claim* pode ser verificado na seção 10.1 da RFC 7519, disponível através do link https://tools.ietf.org/html/rfc7519#section-10.1.
- private: *claims* privados são nomes definidos por uma empresa ou um Authorization Server e que são conhecidos pelas aplicações que os utilizam. Nesse caso, um Resource Server já deve ter conhecimento dos *claims* privados definidos por um Authorization Server. Como exemplo de *claim* privado, podemos usar um atributo que contenha o e-mail do usuário que autorizou a geração do token.

Para que você possa entender o propósito de cada *claim* registrado pela especificação JWT, veja a tabela com o nome e o significado de cada propriedade.

Sigla	Claim	Descrição
iss	Issuer	Indica o criador do token JWT. No nosso caso, foi o Authorization Server da aplicação *bookserver*.
sub	Subject	Identifica para quem o token foi gerado. Geralmente é associado ao Resource Owner que deu a autorização para geração do token. No caso da aplicação *bookserver*, podemos usar o e-mail do Resource Owner.
aud	Audience	Indica quem pode aceitar esse token. Geralmente indica para qual Resource Server o token foi gerado.
exp	Expiration time	Tempo de validade do token em segundos.

Sigla	Claim	Descrição
nbf	Not before	Indica um timestamp de quando o token passará a ser válido.

Sigla	Claim	Descrição
iat	Issued at	Timestamp que informa quando o token foi gerado.
jti	JWT ID	Um identificador único para o token.

Estrutura de um JWT

Agora você já tem uma ideia do que pode estar contido em um token JWT, mas todos esses conceitos ainda podem estar vagos. Como é que podemos representar todos os *claims* de que falamos anteriormente através de um token? O Client agora precisa passar um JSON em vez de uma sequência de caracteres conforme viemos usando até agora? Nada disso.

Um token OAuth 2.0 será sempre opaco para o Client, ou seja, o Client nunca precisará saber nada sobre a estrutura do token ou como ele foi gerado. O Client continuará utilizando um token representado através de uma cadeia de caracteres compacta que pode ser transmitida como parâmetro de URL. Vamos entender como todos esses dados em formato JSON são estruturados em um token representado através de uma simples string.

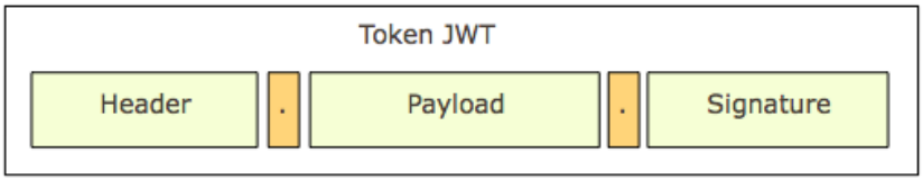

Figura 15.1: Estrutura de um token JWT

A figura anterior mostra a estrutura básica de um token JWT. Conforme podemos ver, um token JWT é formado por três seções separadas por ponto (.), constituídas por um *header*, um *payload* e um *signature*. O *header*, ou cabeçalho, contém informações a respeito do tipo do token e como ele foi assinado ou criptografado.

Mas qual a diferença entre assinar e criptografar um token? Quando assinamos um conteúdo, geramos um hash que serve para garantir a integridade do conteúdo recebido. A especificação JWS (*JSON Web Signature*) define um caso especial de JWT que é usada para definir toda a estrutura de um JWT assinado.

Em contrapartida, quando criptografamos um conteúdo, estamos protegendo os dados de serem lidos por usuários ou sistemas não autorizados. Para o caso de JWT criptografado, devemos contar com a especificação JWE (*JSON Web Encryption*) que também se trata de uma especialização do JWT. Veremos mais detalhes sobre assinatura e criptografia mais adiante.

A questão é que, para definir se um token foi assinado ou criptografado e como isso foi realizado, a especificação JWT conta com outras especificações complementares (JWS, JWE, JWA e JWK) compondo uma coleção conhecida como JOSE (*JavaScript Object Signing & Encryption*). Portanto, é comum encontrar definições do cabeçalho do JWT como **JOSE header**.

Vejamos um exemplo de cabeçalho em que estamos definindo um JWT que não é assinado nem criptografado.

```
{
  "typ": "JWT",
  "alg": "none"
}
```

Como estamos tratando de um JWT não assinado, o atributo alg do cabeçalho foi definido com o valor none.

A segunda seção do token contém o *payload* do JWT. É no *payload* que estão contidos os dados sobre o usuário e sobre o token em si, ou seja, são as *claims* de que falamos anteriormente. Essa estrutura, assim como o cabeçalho, também é representada através de um JSON, conforme vemos a seguir:

```
{
  "iss": "bookserver",
  "sub": "jujuba@mailinator.com",
  "name": "Princesa Jujuba",
  "exp": 1491766663
}
```

A última seção contém a assinatura do *payload*, ou seja, uma string que pode ser usada para verificar a integridade do *payload*. Através de uma assinatura, podemos garantir que uma mensagem não foi alterada no meio do caminho. Isso poderia ser desastroso para um Resource Server, pois ele poderia acabar aceitando um token gerado por qualquer usuário mal-intencionado em vez do Authorization Server.

Já temos todos os dados necessários para compor cada seção de um token JWT, entretanto, para que ele possa ser transmitido através de uma URL, é necessário codificar cada seção em formato JSON com Base 64. Veja a seguir como poderíamos gerar um exemplo de token usando Java puro. Caso queira testar esse trecho de código, importe o projeto testes-jwt que se encontra no diretório livro-spring-oauth2/capitulo-15.

```
public class TesteJwt {
    public static void main(String[] args) {
        String header = new String(
            Base64.getEncoder().encode(getHeaderInBytes()));

        String payload = new String(
            Base64.getEncoder().encode(getPayloadInBytes()));

        String token =  header + "." + payload + ".";

         System.out.println(token);
    }

    private static byte[] getHeaderInBytes() {
        return getFileContentInBytes("header.json");
    }

    private static byte[] getPayloadInBytes() {
        return getFileContentInBytes("payload.json");
    }
    // código omitido
}
```

Considerando o header e o payload que usamos anteriormente, ao executar o código de teste para geração de token, obteremos o seguinte JWT:

ewogICJ0eXAiOiAiSldUIiwKICAiYWxnIjogIm5vbmUiCn0K.ewogICJpc3MiOiAiYm9va3NlcnZlciIsCiAgInN1YiI6ICJqdWp1YmFAbWFpbGluYXRvci5jb20iLAogICJuYW1lIjogIlByaW5jZXNhIEp1anViYSIsCiAgImV4cCI6IDE0OTE3NjY2NjMKfQo=.

Perceba que, para gerar o token, basicamente serializamos o conteúdo usando codificação Base 64 nos permitindo chegar a um conteúdo compacto. No exemplo citado, nosso token se trata de um JWT compactado como JWS.

Assinatura e validação do conteúdo

Observando o token gerado anteriormente, podemos notar que a última seção do JWT não foi gerada, ou seja, não assinamos o payload do token. Qual o problema de não assinar o token?

Como a validação do token não conta mais com uma verificação no banco de dados do Authorization Server, qualquer usuário poderia alterar o payload, e o Resource Server poderia aceitar um token contendo *claims* diferentes das que foram definidas pelo Authorization Server. Dessa forma, seria muito fácil usar um token de outro usuário para realizar operações em seu nome.

A assinatura resolve justamente o problema de integridade do payload. Mas como podemos assinar o payload? E o que significa assinar um conteúdo?

Conforme já falei anteriormente, um token assinado é um tipo específico de JWT conhecido como JWS. Fazendo uma analogia com programação orientada a objetos, o JWT seria uma classe abstrata enquanto que um JWS seria a classe concreta que estende JWT. A especificação do JWS (RFC 7515, que pode ser vista em https://tools.ietf.org/html/rfc7515) define como um token pode ser assinado, bem como a sua forma de serialização.

Analisando a imagem a seguir, podemos notar que uma assinatura é gerada a partir de um algoritmo, uma chave e o próprio conteúdo (*payload*) a ser assinado. A partir de duas entradas, o algoritmo gera uma assinatura que também pode ser chamada de *tag* em algumas literaturas. A tag gerada é usada para compor a terceira seção do token JWS.

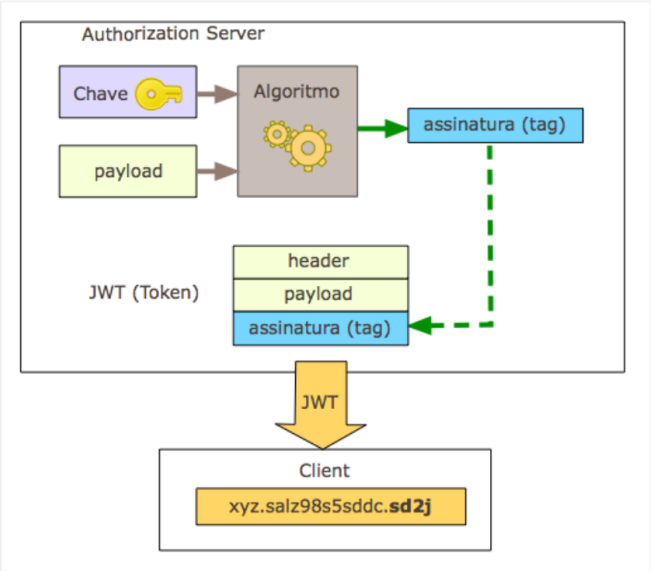

Figura 15.2: Assinatura de um payload

Conforme podemos notar, a assinatura deve ser gerada no Authorization Server que é o componente OAuth que possui tanto os dados para compor o payload quanto a chave de criptografia para assinatura. Em contrapartida, a chave para assinar um conteúdo não deve estar disponível para o Client. Perceba também que o token entregue ao Client continua sendo uma string, que pode ser enviada através do cabeçalho http *Authorization* da mesma forma que fazemos com outros tipos de token.

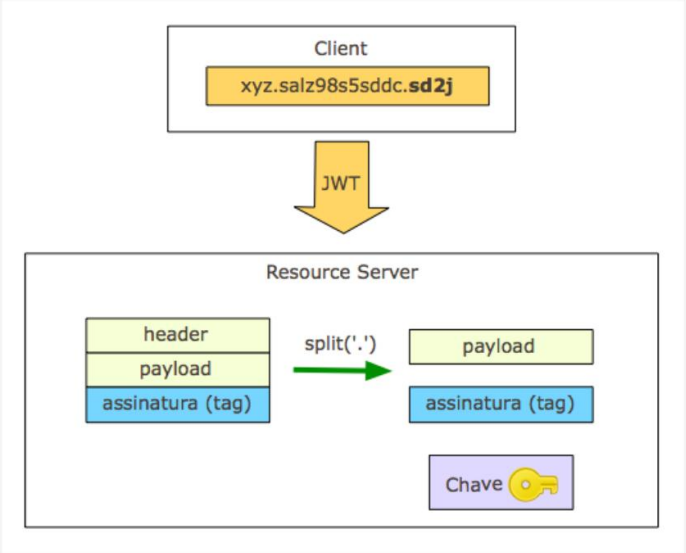

Figura 15.3: Validação do JWT token

Quando um Resource Server recebe um token durante o acesso a algum recurso protegido, ele precisa extrair cada segmento do token JWT para começar a validação. A partir do header extraído, o Resource Server pode identificar qual algoritmo foi utilizado para assinar o token de modo que seja possível aplicá-lo no processo de assinatura realizado pelo Authorization Server. Dessa forma, o Resource Server gera uma nova assinatura e compara com a que foi recebida na terceira seção do token.

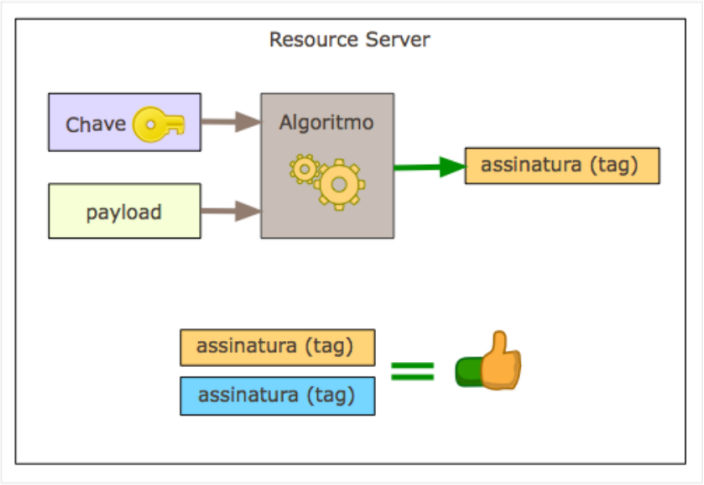

Figura 15.4: Verificação da assinatura JWT do token

Caso as assinaturas sejam iguais, o token pode ser consideradoválido e os dados do *payload* podem ser usados sem perigo.

A questão agora é qual algoritmo utilizar? Os algoritmos possíveis tanto para assinatura quanto para criptografia estão definidos na especificação JWA (*JSON Web Algorithm*). Os algoritmos definidos na especificação JWA podem ser usados para gerar dois tipos de assinatura ou criptografia: simétrica ou assimétrica.

Assinatura simétrica

Ao utilizar um algoritmo de assinatura simétrica, tanto a entidade que gera a assinatura quanto a que valida o conteúdo utilizam uma mesma chave compartilhada. No caso de um OAuth Provider, somente o Authorization Server e o Resource Server possuem acesso às chaves. Para assinar o conteúdo de maneira simétrica, podemos usar funções de criptografia do tipo HMAC (*Hash Based Message Authentication Code*).

MAC e HMAC

A sigla MAC, cujo significado é *Message Authentication Code*, é uma porção de informação que pode ser usada para verificar a autenticidade de um conteúdo bem como a sua integridade. Quando geramos uma assinatura, estamos criando exatamente um valor MAC. Quando falamos de HMAC, ou seja *Hash Based MAC*, estamos tratando de um tipo específico de MAC que é gerado a partir de uma função criptográfica de hash. Mais sobre esse conteúdo pode ser encontrado em https://en.wikipedia.org/wiki/Hash-based_message_authentication_code.

Existem diversos exemplos de funções de criptografia do tipo HMAC e, dentre eles, podemos citar alguns famosos como MD5 e SHA256. Nas imagens anteriores nas quais mostrei como um token pode ser assinado e verificado, tanto o Authorization Server quanto o Resource Server compartilhavam de uma mesma chave para gerar a assinatura. Esse é um exemplo prático da utilização de assinatura simétrica.

Assinatura assimétrica

Quando usamos assinatura assimétrica, o Authorization Server detém um par de chaves, sendo uma pública e outra privada. A chave privada é usada para assinar o token JWT, enquanto que a chave pública é compartilhada com o Resource Server que deve utilizá-la para validar o conteúdo. Dessa forma, apenas o Authorization Server pode assinar um token JWT. Como exemplos de algoritmos de criptografia assimétrica, podemos citar RSA e Elliptic Curve.

Para conhecer um pouco sobre a estrutura dessas chaves e como elas podem ser representadas para o contexto do JWT, visite o link https://mkjwk.org/. Nele é possível gerar um par de chaves para assinatura assimétrica seguindo a especificação JWK (*JSON Web Key*). O JWK nada mais é do que um JSON que define uma estrutura para representar chaves de criptografia de forma padronizada.

Apesar das vantagens de segurança obtidas ao utilizar um algoritmo de assinatura assimétrico, considere o custo de processamento envolvido. O algoritmo RSA é mais complexo, usa mais recursos de processamento e utiliza chaves maiores do que as usadas em algoritmos simétricos. Tudo isso implica no funcionamento da sua aplicação, podendo refletir até mesmo na experiência do usuário.

Criptografia do payload

A criptografia, diferentemente da assinatura, vai proteger contra a leitura não autorizada de um conteúdo, enquanto que a assinatura garante a integridade do conteúdo. Conforme já citado anteriormente, também temos uma especificação para geração de tokens com o payload criptografado. A padronização de como um token pode ser protegido através de criptografia é definida através da especificação JWE, que se trata de outro caso específico de JWT.

Conforme podemos ver na figura a seguir, a estrutura de um token JWE é um pouco diferente da que vimos para o JWS. Em vez de três seções separadas por ponto, agora temos cinco seções:

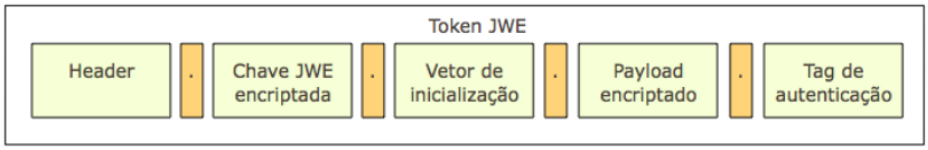

Figura 15.5: Estrutura de um Token JWE

Um token JWE ainda contém o segmento com um cabeçalho (*JOSE header*), entretanto a especificação JWE adiciona mais alguns atributos além dos que já vimos para o JWS. Para se ter uma ideia, o cabeçalho de um token JWE descreve dois tipos de algoritmos de criptografia: um algoritmo para proteção do payload e outro para criptografia da chave usada para criptografar o payload (essa chave compõe a segunda parte do token JWE).

A terceira parte do token JWE pode conter um vetor de inicialização que se trata de uma string gerada randomicamente para adicionar aleatoriedade aos dados criptografados. Em seguida, temos a quarta parte que se trata do payload, e a quinta e última que contém uma chave que permite a verificação da autenticidade do token.

Não entrarei em detalhes sobre como gerar um token JWE, nem sobre os algoritmos envolvidos, pois além de não ser esse o objetivo do livro, tudo isso poderia ser assunto para um capítulo totalmente dedicado (talvez até mesmo um outro livro). Entretanto, quando for necessário utilizar JWE, você pode contar com bibliotecas prontas para a realização dessa tarefa. Uma biblioteca bastante conhecida é a nimbus-jose-jwt, que pode ser melhor estudada através do link https://connect2id.com/products/nimbus-jose-jwt.